CAPH - A bit of history

J. Sérot

October 24, 2014

Abstract

These notes give a short account on how and why the CAPH project started, from an "historical" perspective. They were written after a presentation at the MPP2014 conference [3], where I deliberately put the focus on these aspects in order to illustrate another aspect of the "dataflow revival" trend on which the conference – among others – was surfing. Since it is based on my own personnal experience, it is necessary biased. I hope, however, that it can serve to illustrate the way how advances in the related domains often proceed : by repeated re-discorering and re-cycling of old ideas...

1 The ETCA Dataflow Functional Computer

The origin of the CAPH project can be traced down to years 1990-1993, when I was doing my PhD at the ETCA laboratory[1]. This lab, funded by the DGA[2] was developing experimental perception and computing systems. I've been involved in the development of a massively parallel dataflow computers called the DFC (Dataflow Functional Computer) [4, 5, 6, 7]. This computer, shown on Fig. 1 was built of 1024 custom dataflow processors, physically interconnected to make a 8 x 8 x 16 3D mesh. Its primary goal was to perform real-time, "on-the-fly", image processing on video streams coming from cameras, something which simply cannot be achieved using general purpose processors at this time.

The elementary processor of this machine (called DFP) had been designed by my PhD supervisor, G. Quénot [2] and I was in charge of designing and implementing the software environment for this machine, in order to make it usable by "ordinary" programmers (by ordinary programmers we

[1] Etablissement Technique Central de l'Armement, Arcueil, near Paris.
[2] Délégation Générale de l'Armement, the french DoD

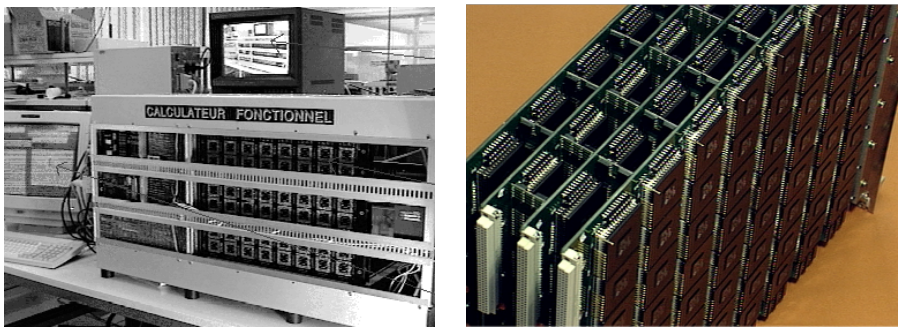

Figure 1: The ETCA Dataflow Functional Computer

meant programmers developping image processing applications but having no special skill in hardware and computer architecture).

From a software point of view, an innovative feature of this machine was the use of a purely functional language, directly inspired from J. Backus' FP [1]. Programs were defined as sets of functional expressions, combining predefined primitives by means of *functional forms*. These expressions were turned to dataflow graphs (DFGs) and these graphs were physically mapped on the mesh of processors. Implementing an algorithm then literally meant "drawing" its DFG on the network (Fig. 2). I personnaly designed and implemented the programming environment that supported by programming model. The main components were a library of primitive operators (running on the DFP), a compiler turning programs written in a dialect of Backus' FP into DFGs and a graphical place-and-route tool for mapping the DFGs onto the 3D mesh[3].

The DFC project proved to be quite successful and several realistic applications, all operating at 25 FPS on B/W or color video streams were implemented. Examples include edge or line extraction, color-based object tracking, connected-component labeling, ...

Indeed, several prototypes of the machine have been built and subsequently used by image processing specialists. But the cost of the realisation and the fact that it was built with ASIC processors – precluded its diffusion outside the lab where it has been designed.

[3]Mapping was initially done manually; then an automatic tool was developed.

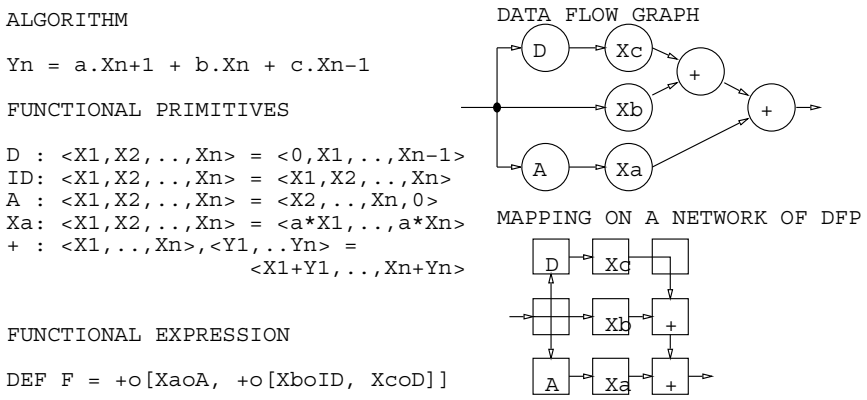

Figure 2: The programming model of the ETCA DFC

2 Smart cameras and VHDL

At the end of 2000s, a colleague of mine, François Berry, started to develop *smart cameras*, in the context of research on embedded perception systems. Smart cameras, such as the ones depicted in Fig 3, associate sensors and on-board processing capabilities in order to provide pre-processed informations rather than raw pixel streams. Such devices have numerous applications, mainly because they alleviate the need of high bandwith communication systems between cameras and remote processing hosts.

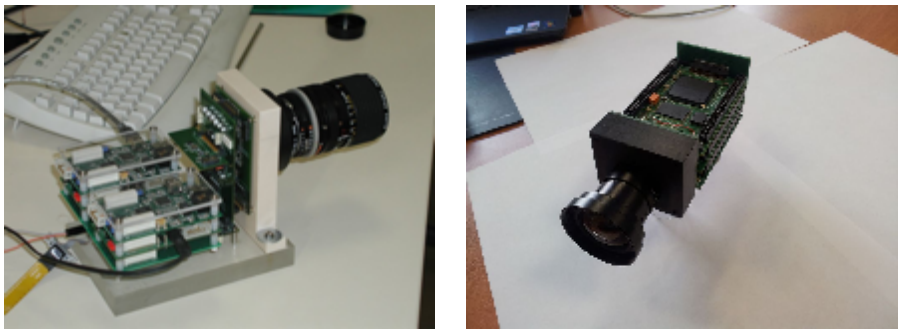

Figure 3: Two smart cameras developed in our lab under the direction of F. Berry

The processing capabilities are generally provided by a micro-processor, a DSP or reconfigurable devices such as FPGAs.

Indeed, FPGAs are a very attractive solution for building smart cameras. Because of the way they are programmed – by specializing behavior of logic elements at the gate level –, they are very well suited to the implementation of algorithms exhibiting a huge amount of fine grain parallelism, which is the case for most of low and mid-level image processing. They can also easily be interfaced to image sensors in order to build direct, on-the-fly, image processions systems

But there's a problem with FPGAs. In the current state-of-the-art, programming them is carried out using *hardware description languages* (HDLs) such as Verilog or VHDL. For a "software" programmer, using these languages is a real challenge because it requires knowledge on hardware design. In particular, many aspects which are completely transparent when dealing with software have to be dealt explicitely : clocks, control signals, *etc.*. So-called *high-level synthesis* tools, especially those offering a C-like front-end, may help but they suffer from limitations which limit their application in practice[4].

My colleague François told me about these programmability issues. At that time, incidentally, I had started a new course on digital system design. So, I really understood his complain about the very low abstraction level of HDLs and the lack of adequate tools for exploiting FPGAs.

By confronting our experiences, we quickly realized that the core of the problem resulted from the gap that existed between the *model of computation* (MoC) that application programmers – those writing image processing applications – are traditionnaly using and the model(s) of computation that can be efficiently implemented on FPGAs.

This just reminded me of the way this problem has actually been solved with the ETCA computer : the gap was *de facto* eliminated by the fact that the dataflow model was used simultanesouly as the programming model and the implementation model. So i had this idea : would be possible to solve the programmability problem by viewing FPGAs the same way we were viewing the 3D mesh of dataflow processors of the DFC, *i.e.* as a physical substrate on which dataflow graphs could be implemented by a simple place-and-route process[5] ? After all, the processor of the DFC was a very simple one[6], so

[4]These limitations mainly stem from the fact that many concepts of the sequential / imperative programming model simply do no map easily/efficiently into hardware.

[5]Afterthought, i wonder whether this idea was not inspired, subconsciously, by the desire to "rebuild" a machine on which i had spent so many exciting hours. I just can't tell exactly. But building on past experiences is definitely a way of learning ... as long as this is not limited to mere replication.

[6]160000 transistors, 1 μm technology.

packing hundreds of them on a moderate size FPGA would certainly be possible.

3 Towards CAPH

Our first attempt at turning this idea into reality was a kind of "reverse engineering" approach. We started by writing a VHDL description of the DFC processor[7] so that we could implement an application described as a dataflow graph on a FPGA by instantiating as many processors as needed, giving them the micro-code to execute to perform the assigned dataflow operation, and connecting them using FIFOs.

But this approach – which really boiled down to "printing" a reduced version of the DFC on a FPGA – had two drawbacks. First, the processors have to be programmed, either by resorting to a predefined library of operators or by writing assembly-level code. Second, and more importantly, it did not take advantage of the full flexibility of FPGAs. When instantiating a processor, for processor, we don't need to allocate hardware resources that won't be used by the function assigned to this processor (for ex, we dont need a multplier in the datapath when computing an addition).

We then realized that we even didn't need the replicate the *structure* of the DFP for implementing dataflow actors since synthetizers actually gave us the possibilty to implement actors at the gate level directly from a behavioral description. The only things that were needed were

- a language allowing the description of the behavior of dataflow actors, which can be easily compiled to RT-level description for synthesis,
- a language for describing the composition of such actors in the form of dataflow graphs, which be compiled as a network of the above connected by FIFOs.

These two requirements were the starting point for the CAPH project[8]. The development really started in 2011 and has continued up to now, fueled by needs and feedbacks I get from the developers of applications to be run, mostly[9] on smart cameras.

[7] This work was carried out by a undergraduate student, xxx

[8] And for me, the opportunity to get back to my favorite research activity, programming language design and implementation, in a very motivating context in which the disciplines of hardware design and programming languages, far from opposing, are actually cross-fertilizing each other.

[9] But not exclusively.

Acknowledgments

I would like to thank all those who directly or indirectly have contributed to the CAPH projet. First, my colleague François Berry. Without his deep knowledge in hardware design, his communicating enthusiasm and skills in finding research funding, none of this would have simply happen. Second, my PhD students who, voluntarily or not, played the role of beta-tester for the CAPH compiler and programming environment, especially Sameer Ahmed and Cédric Bourrasset.

References

[1] J. Backus. Can programming be liberated from the von neumann style?: A functional style and its algebra of programs. *Commun. ACM*, 21(8):613–641, Aug. 1978.

[2] G. Quenot. A data-flow processor for real-time low-level image processing. In *IEEE Custom Integrated Circuits Conference*, pages 12–4, 1991.

[3] J. Sérot and F. Berry. High-level dataflow programming for reconfigurable computing. In *Proc. IEEE 26th International Symposium on Computer Architecture and High Performance Computing*, pages 72–77. IEEE Computer Society, 2014.

[4] J. Sérot, G. M. Quénot, and B. Zavidovique. Functional programming on a data-flow architecture: Applications in real time image processing. *Intl Journal of Machine Vision and Applications*, 7(1):44–56, Dec. 1993.

[5] J. Sérot, G. M. Quénot, and B. Zavidovique. De la programmation fonctionnelle au traitement d'images temps réel. *Technique et Science Informatiques*, 14(7):839–865, Sept. 1995.

[6] J. Sérot, G. M. Quénot, and B. Zavidovique. A visual dataflow programming environment for a real-time parallel vision machine. *Journal of Visual Languages and Computing*, 6:327–347, 1995.

[7] B. Zavidovique, J. Sérot, and G. M. Quénot. Massively parallel dataflow computer dedicated to real time image processing. *Integrated Computer Aided Engineering*, 4(1):9–29, 1997.